A FRAMEWORK FOR PARALLELIZING SAMPLING-BASED MOTION PLANNING ALGORITHMS

A Thesis

by

MATTHEW JAMES BULLUCK

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Nancy Amato
Committee Members,	Lawrence Rauchwerger
	Suman Chakravorty
Head of Department,	Dilma De Silva

December 2017

Major Subject: Computer Science

ABSTRACT

Motion planning is the problem of finding a valid path for a robot from a start position to a goal position. It has many uses such as protein folding and animation. However, motion planning can be slow and take a long time in difficult environments. Parallelization can be used to speed up this process. This research focused on the implementation of a framework for the implementation and testing of Parallel Motion Planning algorithms. Additionally, two methods were implemented to test this framework. The results showed a reasonable amount of speed-up and coverage and connectivity similar to sequential methods.

DEDICATION

To my mother and father, who've helped me through all of this.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Nancy Amato and Lawrence Rauchwerger of the Department of Computer Science and Professor Suman Chakravorty of the Department of Aerospace Engineering.

This was also supported by the work of Shawna Thomas, Timmie Smith, Samuel Jacobs, Jory Denny, Adam Fidel, and Read Sandstrom.

All other work conducted for the thesis was completed by the student independently. No funding was recieved for this research.

NOMENCLATURE

MP	Motion Planning
C-SPACE	Configuration Space
DOF	Degrees of Freedom
kNN	k-Nearest Neighbors
PRM	Probabilistic Roadmap
RRT	Rapidly Exploring Random Tree
STAPL	Standard Template Adaptive Parallel Library

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Motion planning attempts to solve the problem of getting a body from one point to another while avoiding a series of obstacles. Robotic motion planning algorithms are used to compute paths between a given start configuration and a goal configuration in a given environment. Motion planning can be used for a variety of uses such as graphics [1], protein folding [2], and surgical planning [3]. However, for many of these problems the environment can be large and complicated. Additionally, the robot itself can be complex with many degrees of freedom. This leads to problems, which can take a long time for motion planning strategies to solve.

One of the ways to speed this up is to parallelize it. This allows for motion planning algorithms to be on a large number of processors and get more done. As with many of these methods though there are cases of relatively poor scaling due to the overhead of interprocessor communication, bottlenecks within the algorithm itself, or processor load imbalance. Also, these can be very difficult to implement and optimize.

For my thesis, I created a framework from which future Parallel Motion Planning Algorithms can be implemented, run, and tested. This allows for the creation of motion planning strategies, which can compute paths for complex environments quickly given enough processors. To do this, I created a standardized versions of existing algorithms developed in the Parasol Laboratory at Texas A&M which fully utilize the parallelization library, STAPL [4], to allow for efficient parallelization.

2. PRELIMINARIES AND RELATED WORK

2.1 Motion Planning

The robotic motion planning problem can be described as finding a series of configurations between a start and a goal such that these configurations are collision-free. These configurations will form the path which can be used to get from start to goal. To help represent the workspace environment we utilize the concept of C-Space, C-Space is a d dimensional geometric space. Points within space can be classified into two sets C_{free} and C_{obs}. C_{free} represents all configurations within C-space which are considered valid. While C_{obs} represents all configurations which are invalid. Each configuration within this space is represented through a series of d numbers which indicate the entire robot's placement/-pose.

Probabilistic Roadmap(PRM) based Motion planning algorithms can be divided into generating nodes, connecting those nodes, and evaluating the overall roadmap. [5] It is described below in Algorithm 1. The configurations are contained in a graph where each of the nodes is a valid configuration and each edge indicates that there is a valid path between the two configurations. Sampling methods are techniques used to sample the C_{space}. These can be as simple as initially creating a single random sample or can be more complex and take the C_{space} into account to place additional samples. The connection step is made up of two smaller steps neighborhood finding and local planning. The local planner is used to find whether a valid path exists between two given nodes in the environment. It also, can for some implementations, add additional intermediate nodes. Neighborhood finding methods use information about the generated nodes to figure out, between which nodes should local planning be attempted. For this work I'm using a method known as the k nearest neighbors(knn) method. For this, each node will attempt to connect with its k

nearest nodes according to some distance metric where k is defined by the user. Lastly is the evaluation step this is a condition used to figure out whether the algorithm should terminate. Typically, the evaluator will be checking whether there exists a path between some start configuration and the goal configuration. If that condition isn't met the sampling and connections steps will be rerun until it has been met.

Algorithm 1 Probabilistic Roadmap algorithm

```
procedure PRM_METHOD(E, k, n, sampling_method(),
        local_planning_method())
    G ⇐ {, }
    done ← evaluate(G)
    while ¬done do
        G.add_verticies(sampling_method(E))
        for v_i ∈ G.verticies do
            Neighbors ← knn(k, v_i, G.vertices)
            for v_j ∈ Neighbors do
                local_planning_method(v_i, v_j)
            end for
        end for
        done ← evaluate(G)
    end while
end procedure
```

Rapidly-exploring Random Tree(RRT) is another major strategy for solving motion planning problems. [6] This strategy can be divided into the steps of sampling, neighborhood finding, and extending/local planning. This strategy works by initially having some start configuration. Then the planner will generate a random sample within the C-space. The planner will use this sample to steer the direction the tree is generated in.

Algorithm 2 Rapidly Expanding Random Tree(RRT) algorithm

procedure RRT_METHOD($cfg_{init}, dist, num\_iter$)
  $G \Leftarrow \{\{cfg_{init}\},\}$
  $done \leftarrow evaluate(G)$
  while $\neg done$ **do**
    $cfg_{rand} \leftarrow RandomCfg()$
    $cfg_{near} \leftarrow Nearest(cfg_{rand}, G)$
    $cfg_{new} \leftarrow Extend(cfg_{near}, cfg_{rand}, dist)$
    $G.add\_vertex(cfg_{new})$
    $G.add\_edge(cfg_{near}, cfg_{new})$
  end while
end procedure

2.2 Parallel Motion Planning

Parallel motion planning algorithms attempt to distribute tasks between each of the processors. There are multiple steps which can easily be parallelized. [7] Firstly is the node sampling step. Since all of the sampling attempts are independent, this step can be parallelized by distributing each of the sampling attempts between the processors. The next step is parallelizing the neighborhood finding methods. Finding the neighbors of each of the nodes in the roadmap can be done in parallel for the knn method through the use of a map reduce function. Where the map function takes in the node your trying to find the nearest neighbors and another node in the graph, and returns a list containing the distance between the two nodes. The reduce function will then take in two lists of nearest neighbors and return a list of at most length k.

While this can scale well, there is a massive piece of overhead. That overhead is due to inter-processor communication. This is when additional time needs to be spent to have the processors communicate information. The amount of time this eats up can vary heavily between steps and is based off of how much data needs to be communicated and the locality of that data.

Algorithm 3 Parallel PRM algorithm

```
procedure BASICPARALLELPRM(E, k, sampling_method(),
        connection_method())
    G ⇐ {, }
    done ← evaluate(G)
    while ¬done do
        parfor k ← 1..p do
            G.add_verticies(sampling_method(E))
        end parfor
        parfor v_i ∈ G.verticies do
            Neighbors ← parallel_knn(k, v_i)
            for v_j ∈ Neighbors do
                connection_method(v_i, v_j)
            end for
        end parfor
        done ← evaluate(G)
    end while
end procedure
```

Now let's look through the Parallel PRM method to illustrate some of the sources of this interprocessor communication. This is small in the sampling phase, since the only thing that needs to be communicated is how many samples need to be attempted. In the neighborhood finding phase the finding the distance between a single point and each other point in the graph is done in parallel. However; those distances need to be shared between the processors in order to figure out the k nearest neighbors.

Bialkowski et.al goes over a strategy for parallel RRT motion planning. [8] This is done by having some of the most computationally expensive operations, the neighborhood finding and the connection methods, done in parallel. For neighborhood finding this uses a map reduce approach like the one used in the Parallel PRM implementation. However there are some deficiencies in the algorithm. The main thing is that there graphs for measuring the scalability begin when the graph has already had 2000 nodes added to it. This hides what the scalability is like in the beginning of roadmap construction. Unlike the

Parallel PRM approach where a typically large number of nodes are generated via the easily parallelized sampling operation, in each iteration of RRT only one or two nodes gets added. This leads to a situation where in order to achieve full parallelism for the neighborhood finding, at least p iterations must have been run.

Devaurs et.al takes a different approach describing three different schemes which make use of message passing between processors to construct the RRT. The first has each processor generate an RRT until one of the processors reaches a stop condition then it outputs the valid tree. While this does have little interprocessor communication, none of the actual work gets distributed between the processors making it scale poorly. The second method has each of the processors collaboratively construct an RRT. This does distribute the work, but it has a high degree of interprocessor communication. This is since it has each processor communicate node data to each of the processors. This means meaning that the entire tree must be updated on each processor each time a node is added. The third method uses a Manager-work approach to things. The tree is stored exclusively on the manager processor. The manager will generate samples and find the nearest neighbors, while the workers will extend the tree in the direction of the samples. This does distribute work done for extending the graph however the neighborhood finding and sampling step don't scale.

One of the advantages, of using things like parallel neighborhood finding with Radial RRT subdivsion [9] methods, is that since there are $numRegions$ RRTs being created at each iteration $numRegions$ nodes are being created. This means to achieve full parallelism it only takes $p/numRegions$ iterations. However, there's an issue with utilizing subdivision. Firstly, it's necessary to take the structure of the RRT into account. Since the RRT algorithm starts from a single start state the structure of the tree must be taken into account. This is what led to the creation of Radial RRT algorithm. Secondly there are cases where within a region there is a subset of free space which is reachable within the overall roadmap, but isn't reachable within the region. This means that the RRT generated

in that region can't cover that space. This can lead to situations where a path can't be found between the start and goal configurations. Which means the algorithm is probabilistically incomplete. This issue is what led to the creation of the Blind Radial RRT algorithm[10].

2.3 STAPL

These algorithms make use of Parasol's Standard Template Adaptive Parallel Library (STAPL). [11] STAPL is a C++ framework used to provide and create efficient parallel algorithms which are relatively easy to create. It is meant to be a library of Standard C++ components which are similar to the regular STL. This library can also be extended to allow for further functionality. These algorithms can be run on both shared and distributed memory systems. The user frontend is made up of a series of distributed Containers, a set of important Parallel Algorithms, and a set of Views which are used as interfaces for the containers. This relationship is shown in Figure 2.1.

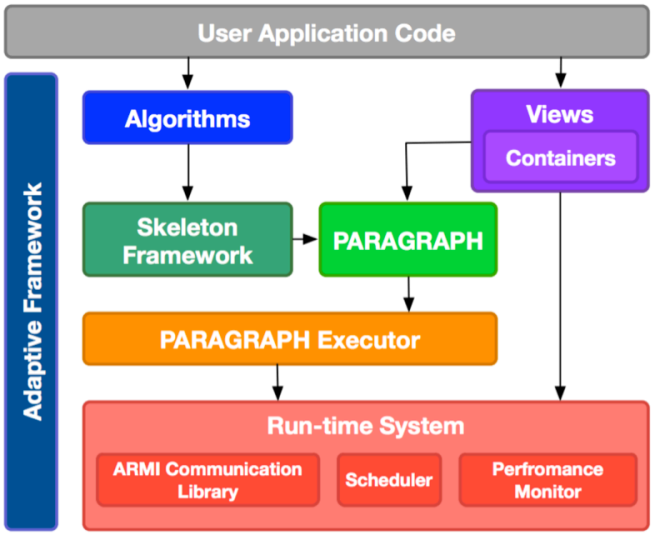

Figure 2.1: Diagram of STAPL Framework

STAPL creates distributed $Containers$. The main data structure PMPL uses from STAPL is its distributed graph data structure[12]. These graphs are made up of a series of vertex and edge descriptors. These allow STAPL to abstract the user from the details of container implementation and focus more on the algorithm development.

In parallel there are two common types of memory management systems, shared and distributed memory systems. In shared memory systems all the processors are able to read and write to the same physical memory. In distributed systems each of the processors has their own memory and in order to share information they need to communicate with one another. For larger clusters of memory distributed memory systems become a necessity due to memory access contention. Because of this most of these tests were run on a distributed memory system.

Interprocessor communication can be reduced using hierarchical graphs. The hierarchical graphs were originally intended for processor networks which have hub node processors which are processors able to connect to a large number of other nodes. This was used to define a data hierarchy within a distributed graph. This hierarchy is used to allow for locale based inter-processor communication where the hub node distributes data to the other processors.

3. METHODS

3.1 Parallel Motion Planning Framework

This project focuses on creating a Parallel Motion Planning Framework that would allow for various methods to be developed, to study performance for those methods, and to find improvements for those methods. To demonstrate the framework's flexibility, I implemented two parallel Motion Planning algorithms for it. These are Parallel PRM and Subdivision-based PRM. For this I have modified parts of the Probabilistic Motion Planning Library (PMPL) and STAPL. This is shown in Figure 3.1, which shows a diagram of the Parallel Framework along with the two implemented methods.

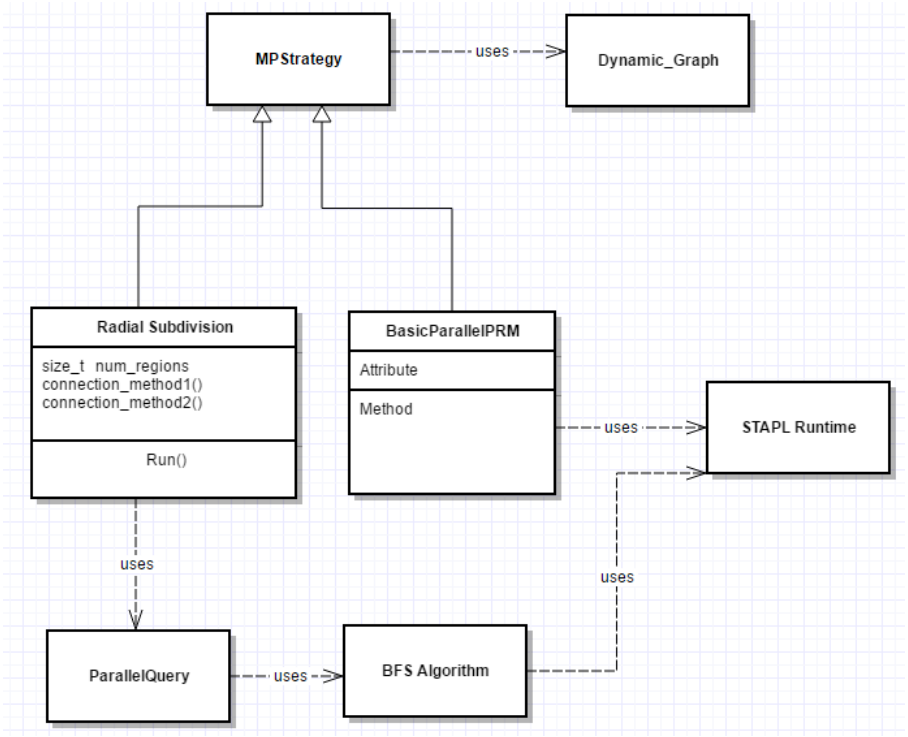

Figure 3.1: Diagram of Parallel MP Framework

3.2 Parallel PRM

For PMPL I added a functional Parallel PRM method as described in Algorithm 3. However, the version used here is slightly different. To limit duplicate work, after generating samples the algorithm will now collect all the new node information and limit connection attempts to those newly added nodes. This massively decreases unnecessary work. However, it does increase interprocessor communication by a significant degree. While there was a previous implementation, it didn't have any parallelization in the connection function and wasn't fully utilizing STAPLs feature set.

I also made a parallel query function which makes use of STAPL's parallel Breadth-First Search algorithm to check for paths from some start to goal in parallel. To do this I had to make some small modifications to the BFS algorithm. Previously the parallel MP strategies only were only able to use a conditional evaluator which terminated the strategy when a certain number of nodes or edges had been generated. Now it can check whether there is a path between start and goal.

3.3 Parallel PRM using Subdivision

I designed and implemented subdivision based methods similar in spirit to methods which were originally designed by Sam Jacobs[13] [9]. This is described below in Algorithm 4. Subdivision works by first having the program divide the environment into a number of self-contained regions. Each of these regions have their own samples and motion planning methods. They will each run that methods within their respective region. Once this is done the planner will connect samples between adjacent regions in parallel. The purpose of this is to lower the necessary amount of interprocessor communication by utilizing locality information to lower the number of nodes checked against for neighborhood finding. One of the main considerations is how to divide the environment. This can massively effect the runtime of the algorithms, since a poor division of the environment

can lead to work being distributed unevenly. This can lead to significant bottlenecks. Currently we have the algorithm evenly divide the environment into a user specified number of regions on a grid. Other subdivision methods are possible, but they aren't implemented here.

Originally this method was somewhat restrictive and only allowed for a single processor to do work for each of the regions. I make use of a Hierarchical Graph View to manage subdivision. This Hierarchical graph will be a set of super vertices indicating the relative location of each of the regions to be sampled in. These super vertices are connected according to region adjacency. While implementing this multiple neighborhood finding methods were tested to the best for inter-region connection. The best one that was found was the k-closest pairs method.

Algorithm 4 Regular Subdivision Algorithm

procedure REGULARSUBDIVISION($E, num\_regions, sampling\_method(), connection\_method1(), connection\_method2()$)
  $RegionMap \leftarrow Decompose(E, num\_regions)$
  $G \leftarrow \{,\}$
  $G' \leftarrow HierarchicalPartition(G, RegionMap)$
  $done \leftarrow evaluate(G)$
  while $\neg done$ **do**
    parfor $r \in RegionMap.verticies$ **do**
      $g' \leftarrow G'.get\_super\_vertex(r)$
      $S \leftarrow Sample(r.boundary, sampling\_method())$
      $g'.Add(S)$
      $Connect(S, g'.descriptors, connection\_method1())$
    end parfor
    parfor $edge(g'_i, g'_j) \in G'.SuperGraphEdges$ **do**
      $S_i \leftarrow g'_i.descriptors$
      $S_j \leftarrow g'_j.descriptors$
      $Connect(S_i, S_j, connection\_method2())$
    end parfor
    $done \leftarrow evaluate(G)$
  end while
end procedure

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

The library was compiled using the gcc c++ compiler version 4.9.2. Each of the runs were performed on a Cray Super Computer at Texas A&M University. These runs used up to 512 processors in counts increasing exponentially by powers of 2. For each processor count 10 runs were found. The environment the robot planned on was a 3-d cluttered environment. These used uniform random sampling and k-nearest neighbors neighborhood finding with k=5. The inter-region connection used by the regular subdivision method was k-closest pairs with k=10. These were run for a fixed number of nodes. These methods

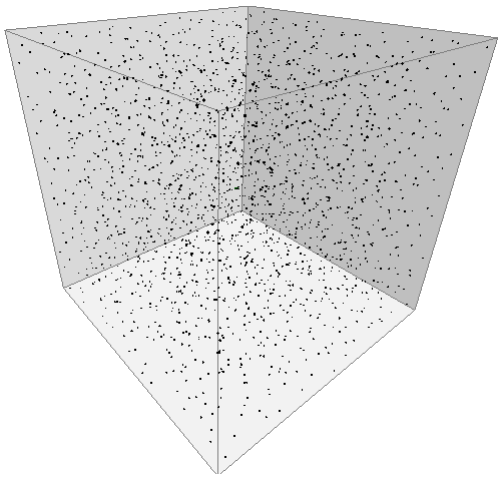

Figure 4.1: 3D Cluttered Environment

were run in the environment shown in Figure 4.1. It is a large cluttered environment with 1000 obstacles placed randomly throughout the environment.

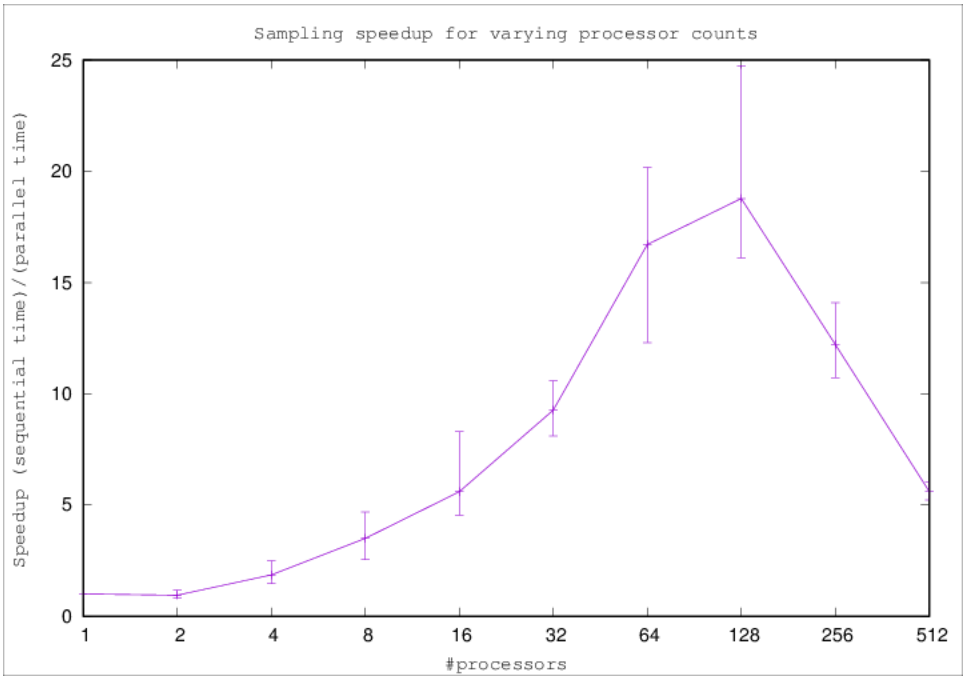

Figure 4.2: Scalability of Sampling in Parallel PRM

4.2 Parallel Performance

These tests will be measuring the speedup for generating samples, neighborhood finding and connecting nodes. They will be performed on the Parallel PRM and Regular Subdivision motion planning methods.

4.2.1 Parallel PRM Results

Figure 4.2 shows how well the Sampling function scales with increasing numbers of processors. It demonstrates that the sampling function of the Parallel PRM method doesn't appear to scale well. The main reason for that has to do with the way in which the sampling function works. To save on unnecessary work, in each iteration of the PRM algorithm, connection is only performed on the nodes which have most recently been added to the parallel graph. The sampling function needs to communicate the ids of the most recently

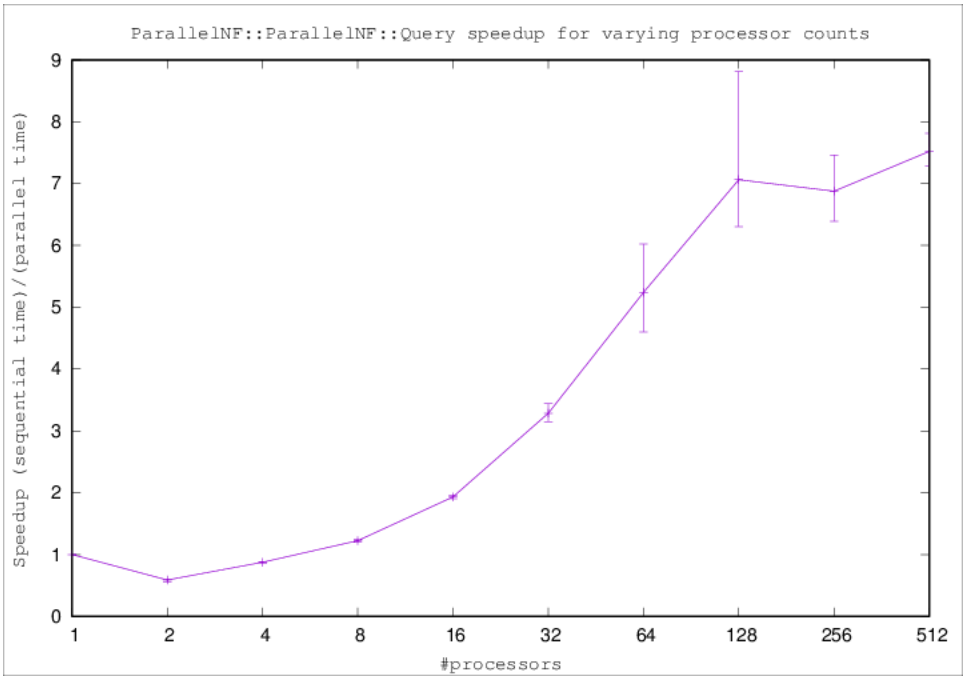

Figure 4.3: Scalability of Neighborhood Finding in Parallel PRM

added vertices to each of the functions. So that additional interprocessor communication is what is causing the poor scalability.

Figure 4.3 shows the scalability of the Neighborhood Finding method in Parallel PRM. As you can see put the scalability is poor. However, that expected considering the amount of interprocessor communication necessary.

Figure 4.4 shows how well the Local Planning scales in Parallel PRM. Overall it scales fairly well. However not perfectly. This is because before Local Planning, the method checks whether the two nodes are already connected. If they are then attempting Local Planning is unnecessary so it isn't performed. However, whether that connection exists or not may not be processor local and as such inter processor communication is necessary. This is optimized so that the time spent waiting is lessened. However, it still influenced the performance and is why the scalability is suboptimal.

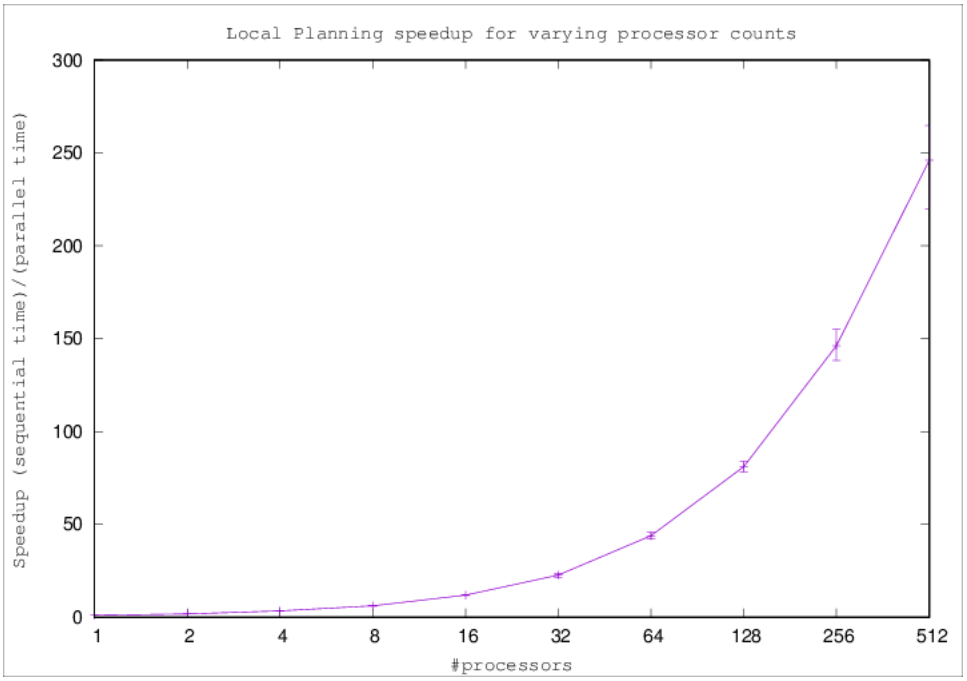

Figure 4.4: Scalability of Local Planning in Parallel PRM

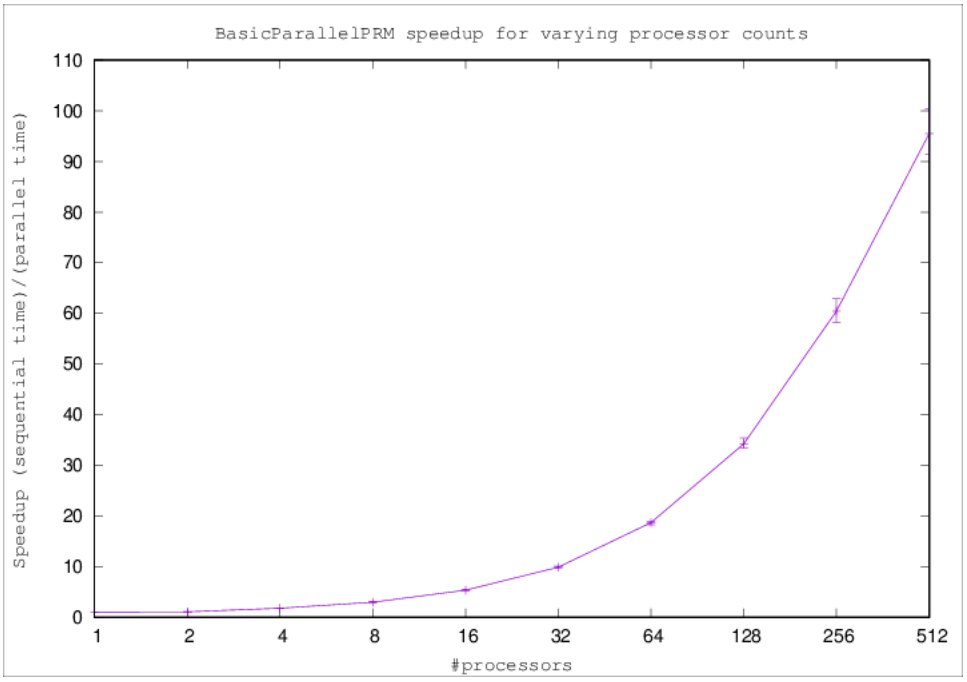

Figure 4.5: Overall Scalability of Parallel PRM

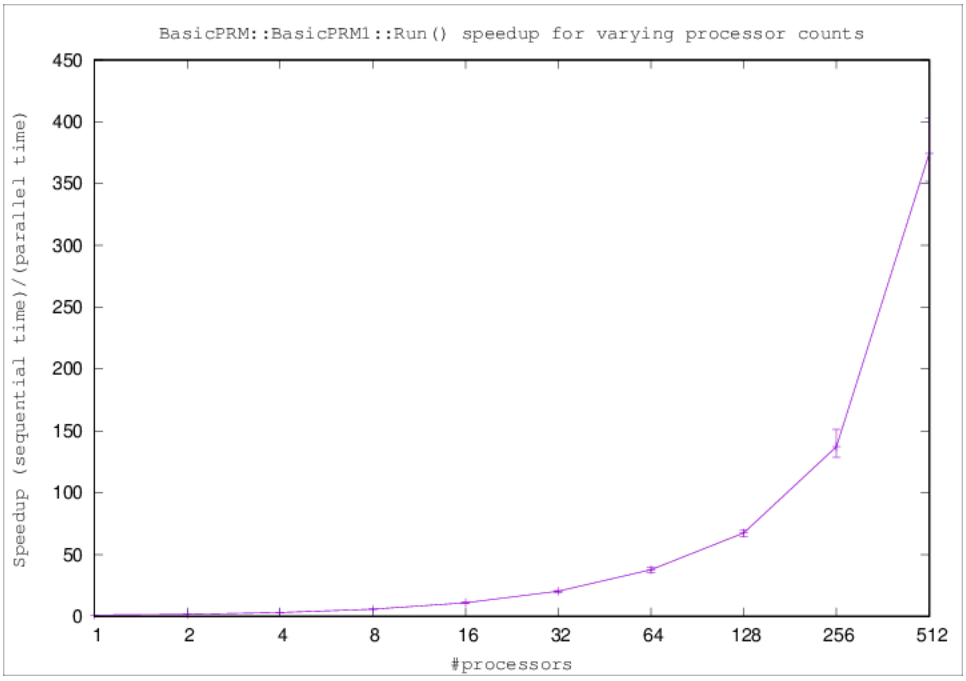

Figure 4.6: Scalability of the BasicPRM in each region

Figure 4.5 shows the how well Parallel PRM scales overall. It scales well all things considered. It's being hindered by the overhead of neighborhood finding. However; there's not much that can be done about it considering the high amount of interprocessor communication in the Neighborhood Finding method.

4.2.2 Subdivision Results

Figure 4.6 shows how the intra-region sequential PRM algorithm, that is run at each iteration of the Regular Subdivision, scales. Overall it scales fairly well. While it's not perfect this is because while the number of nodes does get evenly divided that doesn't necessarily mean that the work gets evenly divided. This is because certain regions require more work than other regions.

Figure 4.7 shows that the neighborhood finding within the region scales at a quadratic rate. This is because whenever the regions are divided in half there are half as many nodes

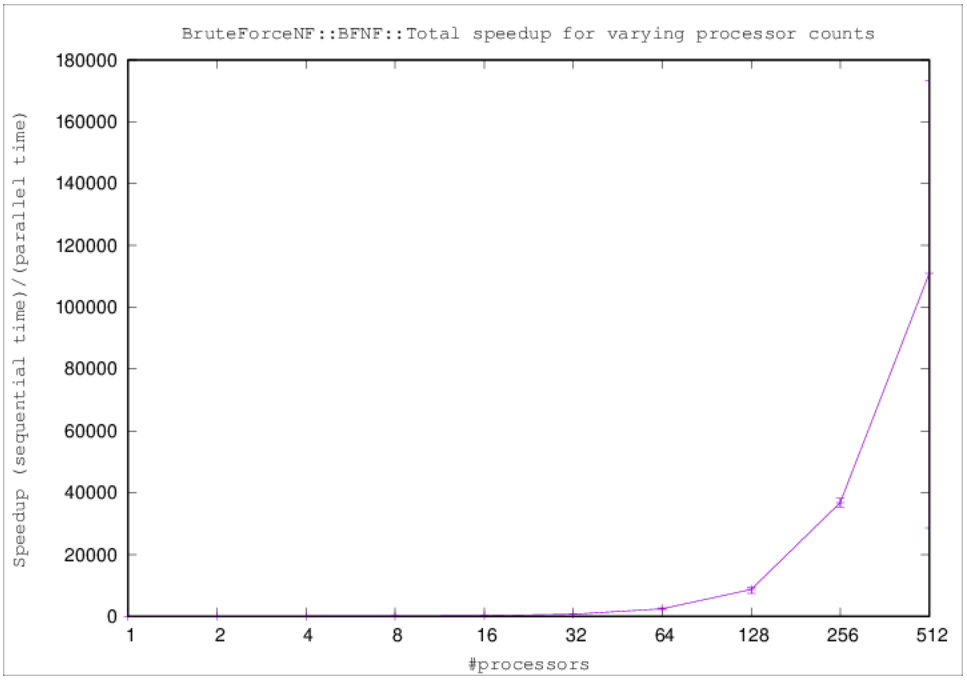

Figure 4.7: Scalability of Intra-Region Neighborhood Finding

in the region. This means half as many nodes to check against when finding neighbors and also, half as many calls to the neighborhood finder.

Figure 4.8 shows the speedup in Connection time. This shows very good speedup. Which is good because it takes up the most runtime. The reason for this is because the number of nodes it must attempt connections for scales with the number of processors.

Figure 4.9 the scalability for generating nodes within the region is poor. That might seem odd at first because the number of nodes generated is evenly divided between each of the processors. However, the reason for this is because even though the nodes themselves are evenly divided this doesn't necessarily mean that the work winds up being divided. This is because specific node generations require a greater number of attempts than others. This means that certain processors will take more time generating samples than others. Since this is measuring the processor with the longest node generation time there will be

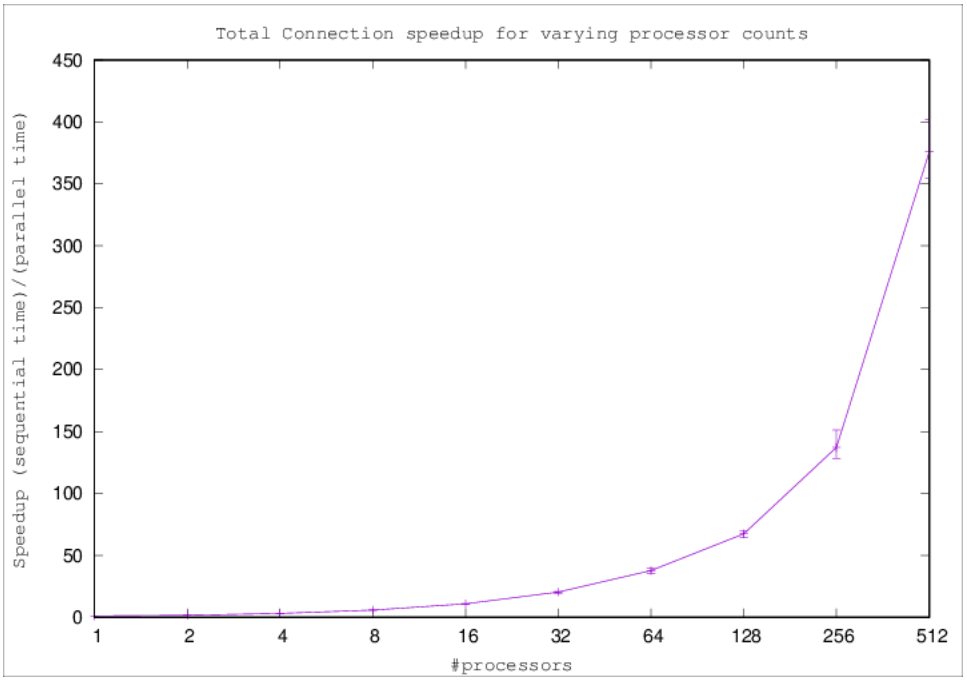

Figure 4.8: Scalability for Intra-Region Connection Time

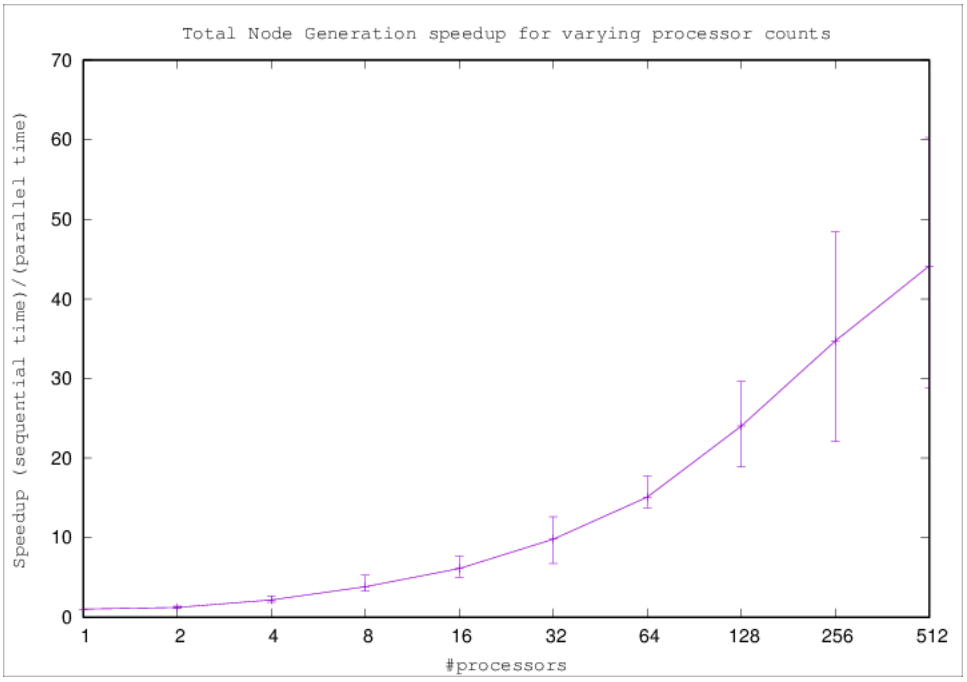

Figure 4.9: Scalability for Node Generation

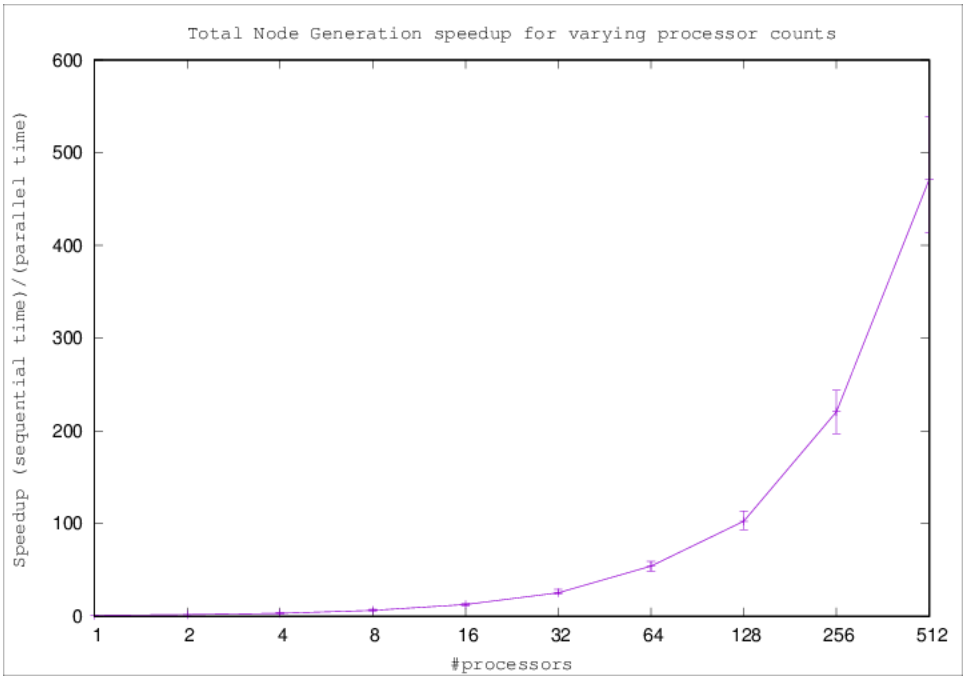

Figure 4.10: Scalability for Node Generation using the average time between processors

some processors that take longer. Figure 4.10 helps illustrate this issue. As you can see that average between the processors does scale well. Meaning that yes, the nodes generated does get distributed, but the work to generate these nodes doesn't. Also, there are some cases where entire region is in $C_{obstacle}$ space meaning that it will attempt to generate more nodes than the other processors until it terminates. However; that was never the case with these results at least. Luckily this doesn't have a major effect on the overall runtime of the BasicPRM method because of how relatively little time Generating nodes takes.

Both Figures 4.11 and 4.12 show the runtime of the methods as opposed to the scalability for them the reason for this is because these are never used for single processor runs. This means that it's a better idea to show the runtime. Figure 4.11 shows the interestingly the time for this decreases as the number. The reason for this is that much like the intra-region neighborhood finding as the number of nodes decreases both the number of calls

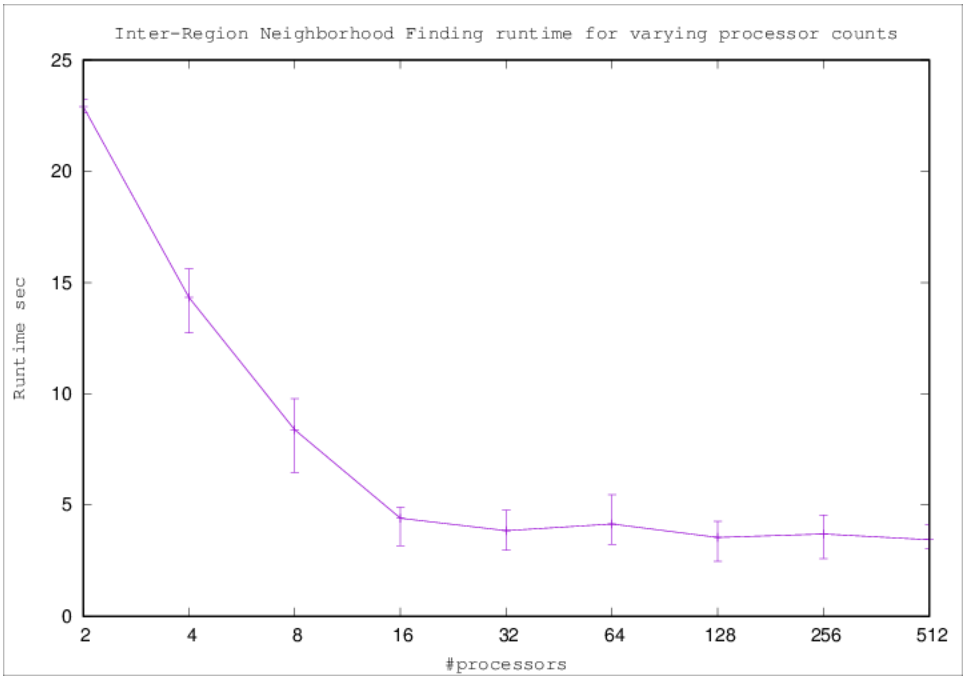

Figure 4.11: Runtime for Neighborhood Finding between regions

and the number of nodes to check against for those calls decreases. This would normally give a quadratic speedup; however due to the high degree of inter-processor communication necessary for this step the speedup is heavily limited.

Figure 4.12 shows the amount of time for local planning between regions. Interestingly it doesn't seem to scale as well as it should. After all most of the inter-region communication is obtained during the Neighborhood Finding attempts. That means that this should speed up. However, the reason it doesn't scale is because there is one major piece of inter-processor communication that happens here. And that is adding the edges between the non-processor local vertex and the processor local vertex.

Lastly, Figure 4.13 shows the overall runtime of the region connection. This is just a combination of the previous two graphs so there's not much to say about this. Just that there appears to be a sweet spot where the runtime is minimized at around 16 processors.

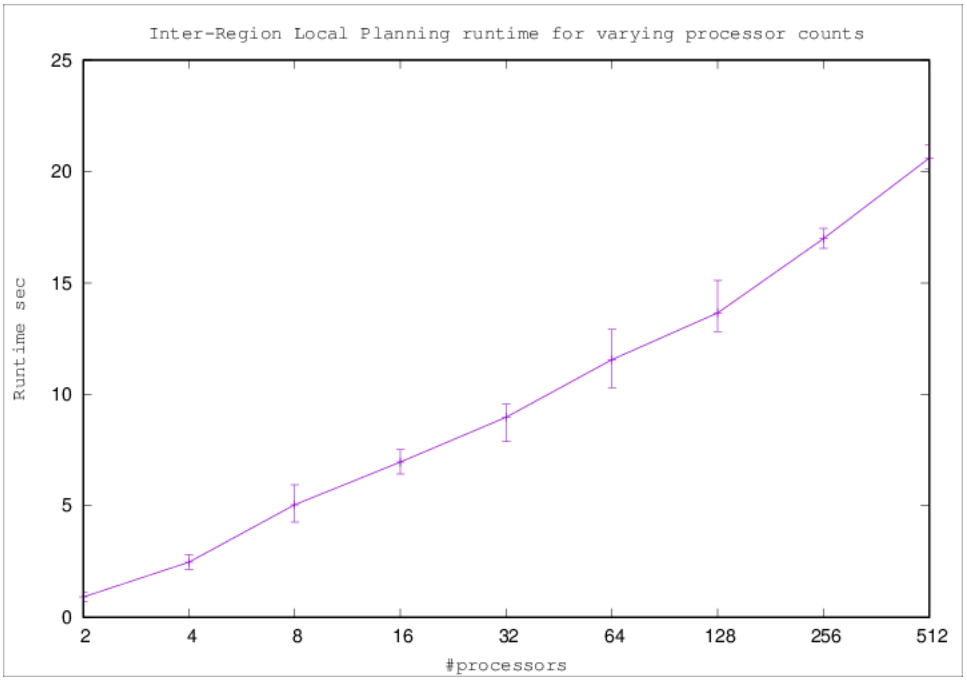

Figure 4.12: Runtime for Local Planning between regions

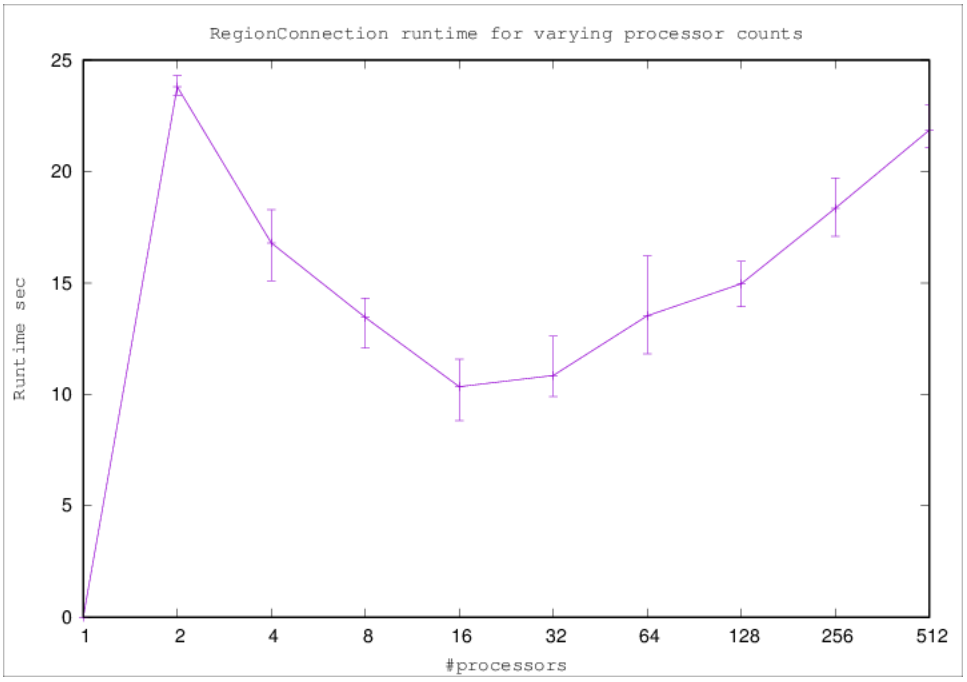

Figure 4.13: Runtime for Region connection overall

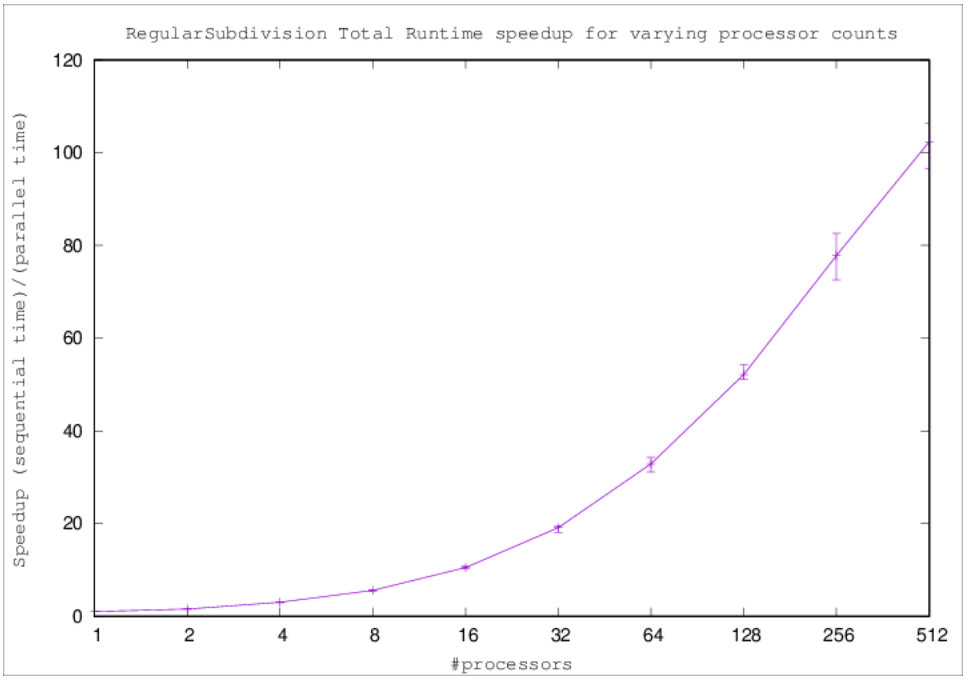

Figure 4.14: Scalability of the Entire Subdivision Method

Lastly Figure 4.14 shows the overall scalability of the entire subdivision method. As you can see this is scaling poorly. It's better scalability than Parallel PRM, but the method still has room for improvement. The reason for this can be seen if you look back at the inter-region connection time. The inter-region connection time doesn't scale and because the overall time necessary for it is relatively high. This limits the amount of scalability that can be achieved for this method.

4.3 Motion Planning Performance

This will measure the coverage by generating samples within the environment and finding the percentage of those that can be connected to the roadmap. This is important because it demonstrates how much of an effect using these methods has on the generated roadmap. These tests also make use of the Grid Maze Environment shown in figure 4.3 in order to verify their results.

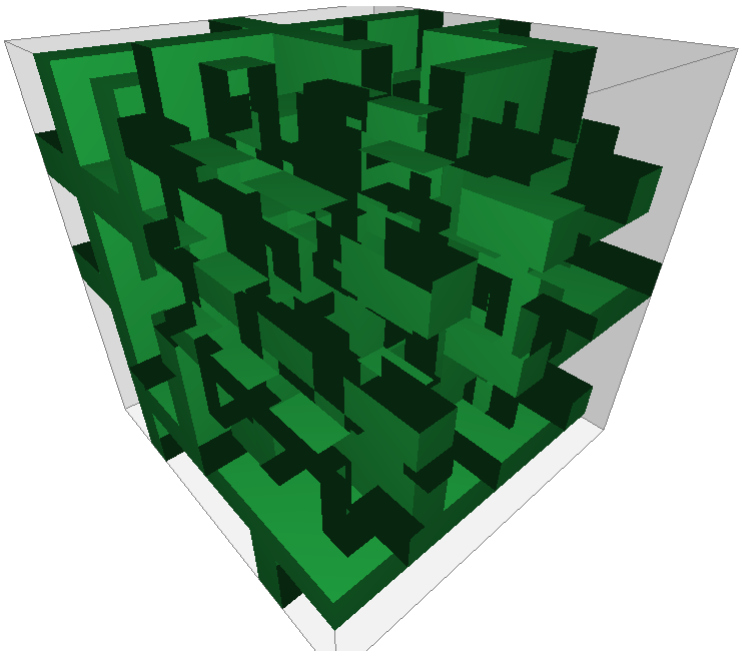

Figure 4.15: 3D Grid Maze Environment

4.3.1 Parallel PRM Results

Table 4.1: Parallel PRM Connectivity and Coverage

Test	Coverage	Connectivity
Sequential PRM in Cluttered Env	100%	100%
Basic Parallel PRM in Cluttered Env	100%	100%
Sequential PRM in 3D Grid Maze Env	95.5%	91.18%
Basic Parallel PRM in 3D Grid Maze Env	94.23%	91.28%

Table 4.1 compares the coverage and connectivity between Basic Parallel PRM and Sequential PRM. Each of the runs on the cluttered environment had a 100% coverage and connectivity rate for all processor counts. However, this could just be due to the

high number of samples or it could be that my environment is too simple. To check the performance, additional tests were run which, compared the results of 10 64 processor runs on the 3D Grid Maze environment versus 10 sequential runs on the same environment. Additionally, the number of samples was lowered to only be 1000. The sequential runs had an average coverage of 95.5% and a connectivity rate of 91.18%. The 64 processor runs had an average coverage of 94.23% and an average connectivity rate of 91.28%. As you can see these are incredibly close. This makes sense considering Parallel PRM performs all the same steps of regular PRM it just performs those steps in parallel.

4.3.2 Subdivision Results

Table 4.2: Regular Subdivision Connectivity and Coverage

Test	Coverage	Connectivity
Sequential PRM in Cluttered Env	100%	100%
Subdivision PRM in Cluttered Env	100%	100%
Sequential PRM in 3D Grid Maze Env	95.5%	91.18%
Subdivision PRM in 3D Grid Maze Env	94.9%	90.08%

Table 4.2 compares the coverage and connectivity between Subdivision PRM and Sequential PRM. Much like the previous results there was 100% coverage and connectivity so like last time additional tests were performed on the more complex 3D maze environment. Like the previous section there were tests comprised of 10 sequential runs and 10 64 Processor runs. For the 64 processor runs the environment was divided into a set of 4x4x4 boxes. On average the single processor runs had a coverage rate of 95.5% and a connectivity rate of 91.18%. Meanwhile; the 64 processor count runs had an average Coverage rate of 94.9% and an average Connectivity rate of 90.08%. While the subdivision method had worse connectivity and coverage, it's still comparable and within some margin of error.

5. CONCLUSION

This project created a framework which allows for making Parallel sampling based motion planning algorithms. It will allow for the creation of scalable parallel motion planning strategies. This will be made up of a series of standardized existing algorithms including, Parallel PRM, Regular Subdivision, Radial RRT, and Radial Blind RRT. This will also include parallel neighborhood finding, connection, and roadmap evaluation components which future algorithms can make use of.

Future work for this work would largely focus on improvements to the Subdivision Method. Currently the subdivision method is dividing the environment arbitrarily. For these a different method would have to be set up to measure the effects of these changes. This is because while they wouldn't necessarily improve scalability or Coverage/Connectivity. They would reduce the amount of time necessary to get good Coverage and Connectivity, while at least maintaining scalability if not improving it. Also, currently the Subdivision method checks for collisions with each obstacle in the environment. This includes irrelevant obstacles, such as ones the robot couldn't possibly collide with while being within the region. While I originally planned on implementing the RadialRRT method, I didn't have enough time to do so. However, with the framework I've set up doing so won't be too difficult. Many of the steps, such as region division and connection can be repurposed or are already included in the Regular Subdivision method.

REFERENCES

[1] O. B. Bayazit, J.-M. Lien, and N. M. Amato, "Roadmap-based flocking for complex environments," in *Computer Graphics and Applications, 2002. Proceedings. 10th Pacific Conference on*, pp. 104–113, IEEE, 2002.

[2] N. M. Amato and G. Song, "Using motion planning to study protein folding pathways," *Journal of Computational Biology*, vol. 9, no. 2, pp. 149–168, 2002.

[3] J.-C. Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *The International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999.

[4] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library (stapl)," in *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '98, (London, UK, UK), pp. 402–409, Springer-Verlag, 1998.

[5] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[6] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," tech. rep., Iowa State University, Department of Computer Science, 1998.

[7] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, vol. 1, pp. 688–694, IEEE, 1999.

[8] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the rrt and the rrt," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*,

pp. 3513–3518, Sept 2011.

[9] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed rrt for motion planning," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 5088–5095, IEEE, 2013.

[10] C. Rodriguez, J. Denny, S. A. Jacobs, S. Thomas, and N. M. Amato, "Blind rrt: A probabilistically complete distributed rrt," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1758–1765, IEEE, 2013.

[11] G. Tanase, A. Buss, A. Fidel, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, *et al.*, *The STAPL parallel container framework*, vol. 46. ACM, 2011.

[12] A. Fidel, N. M. Amato, L. Rauchwerger, *et al.*, "The stapl parallel graph library," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 46–60, Springer, 2012.

[13] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, "A scalable method for parallelizing sampling-based motion planning algorithms," in *In Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pp. 2529–2536, 2012.